3 Your first application

This chapter covers

- Creating your first ASP.NET Core web application
- Running your application
- Understanding the components of your application

In the previous chapters, I gave you an overview of how ASP.NET Core applications work and when you should use them. Now you should set up a development environment to use for building applications.

> **TIP** See appendix A for guidance on installing the .NET 7 software development kit (SDK) and choosing an editor/integrated development environment (IDE) for building ASP.NET Core apps.

In this chapter, you'll dive right in by creating your first web app. You'll get to kick the tires and poke around a little to get a feel for how it works. In later chapters, I'll show you how to go about customizing and building your own applications.

As you work through this chapter, you should begin to get a grasp of the various components that make up an ASP.NET Core application, as well as an understanding of the general application-building process. Most applications you create will start from a similar template, so it's a good idea to get familiar with the setup as soon as possible.

DEFINITION A *template* provides the basic code required to build an application. You can use a template as the starting point for building your own apps.

I'll start by showing you how to create a basic ASP.NET Core application using one of the Visual Studio templates. If you're using other tooling, such as the .NET command-line interface (CLI), you'll have similar templates available. I use Visual Studio 2022 and ASP.NET Core 7 with .NET 7 in this chapter, but I also provide tips for working with the .NET CLI.

TIP You can view the application code for this chapter in the GitHub repository for the book at http://mng.bz/5wj1.

After you've created your application, I'll show you how to restore all the necessary dependencies, compile your application, and run it to see the output. The application will be simple, containing the bare bones of an ASP.NET Core application that responds with `"Hello World!"`

Having run your application, your next step is understanding what's going on! We'll take a journey through the ASP.NET Core application, looking at each file in the template in turn. You'll get a feel for how an ASP.NET Core application is laid out and see what the C# code for the smallest possible app looks like.

As a final twist, you'll see how to extend your application to handle requests for static files, as well as how to create a simple API that returns data in standard JavaScript Object Notation (JSON) format.

At this stage, don't worry if you find parts of the project confusing or complicated; you'll be exploring each section in detail as you move through the book. By the end of the chapter, you should have a basic understanding of how ASP.NET Core applications are put together, from when your application is first run to when a response is generated. Before we begin, though, we'll review how ASP.NET Core applications handle requests.

3.1 A brief overview of an ASP.NET Core application

In chapter 1, I described how a browser makes an HTTP request to a server and receives a response, which it uses to render HTML on the page. ASP.NET Core allows you to generate that HTML dynamically depending on the particulars of the request, so that (for example) you can display different data depending on the current logged-in user.

Suppose that you want to create a web app to display information about your company. You could create a simple ASP.NET Core app to achieve this goal; later, you could add dynamic features to your app. Figure 3.1 shows how the application would handle a request for a page in your application.

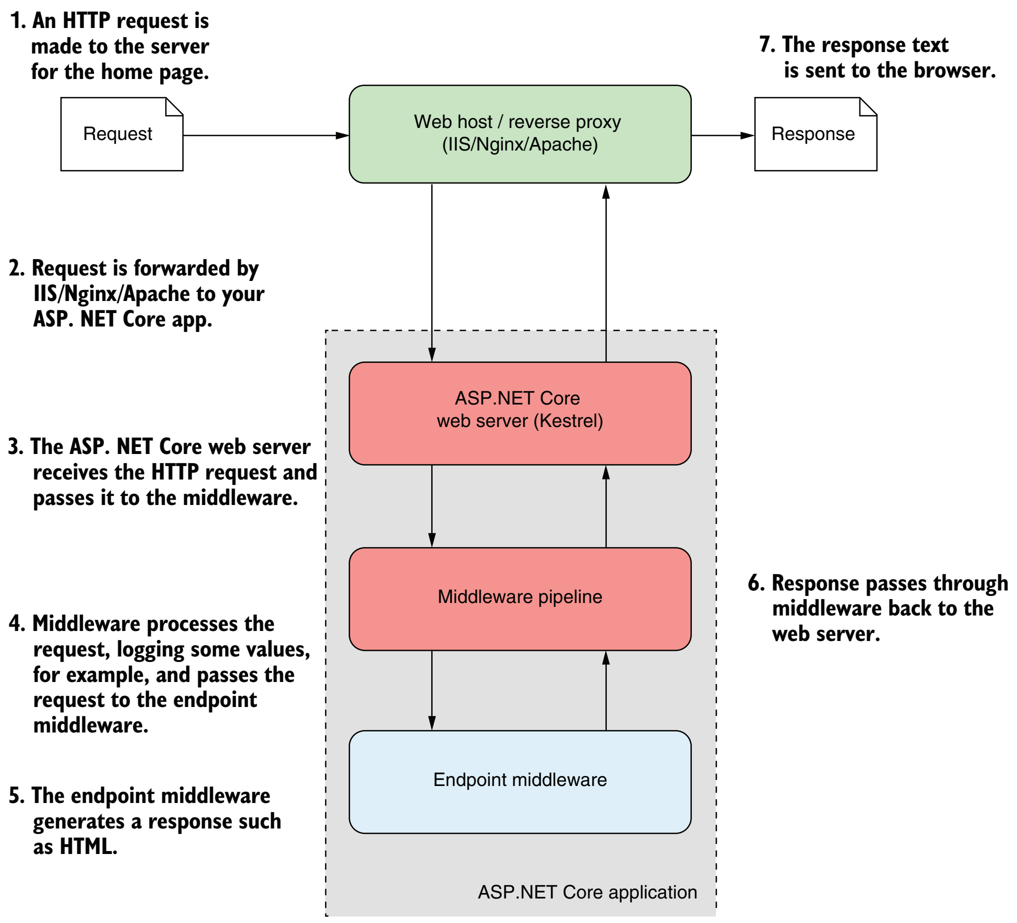

1. An HTTP request is made to the server for the home page.
Request
Web host / reverse proxy (IIS/Nginx/Apache)
7. The response text is sent to the browser.
Response
2. Request is forwarded by IIS/Nginx/Apache to your ASP. NET Core app.
ASP.NET Core web server (Kestrel)
3. The ASP. NET Core web server receives the HTTP request and passes it to the middleware.
Middleware pipeline
6. Response passes through middleware back to the web server.
4. Middleware processes the request, logging some values, for example, and passes the request to the endpoint middleware.
Endpoint middleware
5. The endpoint middleware generates a response such as HTML.
ASP.NET Core application

Figure 3.1 An overview of an ASP.NET Core application. The ASP.NET Core application receives an incoming HTTP request from the browser. Every request passes to the middleware pipeline, which potentially modifies it and then passes it to the endpoint middleware at the end of the pipeline to generate a response. The response passes back through the middleware to the server and finally out to the browser.

Much of this diagram should be familiar to you from figure 1.3 in chapter 1; the request and response and the ASP.NET Core web server are still there. But you'll notice that I've added a reverse proxy to show a common deployment pattern for ASP.NET Core applications. I've also expanded the ASP.NET Core application itself to show the middleware pipeline and the endpoint middleware—the main custom part of your app that goes into generating the response from a request.

The first port of call after the reverse proxy forwards a request is the ASP.NET Core web server, which is the default cross-platform Kestrel server. Kestrel takes the raw incoming network request and uses it to generate an `HttpContext` object that the rest of the application can use.

The HttpContext object

The `HttpContext` constructed by the ASP.NET Core web server is used by the application as a sort of storage box for a single request. Anything that's specific to this particular request and the subsequent response can be associated with it and stored in it, such as properties of the request, request-specific services, data that's been loaded, or errors that have occurred. The web server fills the initial `HttpContext` with details of the original HTTP request and other configuration details and then passes it on to the rest of the application.

NOTE Kestrel isn't the only HTTP server available in ASP.NET Core, but it's the most performant and is cross-platform. I'll refer only to Kestrel throughout the book. A different web server, IIS HTTP Server, is used when running in-process in *Internet Information Services* (IIS). The main alternative, HTTP.sys, runs only in Windows and can't be used with IIS.[1]

Kestrel is responsible for receiving the request data and constructing a .NET representation of the request, but it doesn't attempt to generate a response directly. For that task, Kestrel hands the `HttpContext` to the middleware pipeline in every ASP.NET Core application. This pipeline is a series of components that process the incoming request to perform common operations such as logging, handling exceptions, and serving static files.

NOTE You'll learn about the middleware pipeline in detail in chapter 4.

At the end of the middleware pipeline is the *endpoint* middleware, which is responsible for calling the code that generates the final response. In most applications that code will be a Model-View-Controller (MVC), Razor Pages, or minimal API endpoint.

Most ASP.NET Core applications follow this basic architecture, and the example in this chapter is no different. First, you'll see how to create and run your application; then you'll look at how the code corresponds to the outline in figure 3.1. Without further ado, let's create an application!

3.2 Creating your first ASP.NET Core application

In this section you're going to create a minimal API application that returns `"Hello World!"` when you call the HTTP API. This application is about the simplest ASP.NET Core application you can create, but it demonstrates many of the fundamental concepts of building and running applications with .NET.

You can start building applications with ASP.NET Core in many ways, depending on the tools and operating system you're using. Each set of tools has slightly different templates, but the templates have many similarities. The example used throughout

[1] If you want to learn more about Kestrel, IIS HTTP Server, and HTTP.sys, this documentation describes the differences among them: http://mng.bz/6DgD.

this chapter is based on a Visual Studio 2022 template, but you can easily follow along with templates from the .NET CLI or Visual Studio for Mac.

NOTE As a reminder, I use Visual Studio 2022 and ASP.NET Core with .NET 7 throughout the book.

Getting an application up and running locally typically involves four basic steps, which we'll work through in this chapter:

1. *Generate*—Create the base application from a template to get started.
2. *Restore*—Restore all the packages and dependencies to the local project folder using NuGet.
3. *Build*—Compile the application, and generate all the necessary artifacts.
4. *Run*—Run the compiled application.

Visual Studio and the .NET CLI include many ASP.NET Core templates for building different types of applications, such as

- *Minimal API applications*—HTTP API applications that return data in JSON format, which can be consumed by single-page applications (SPAs) and mobile apps. They're typically used in conjunction with client-side applications such as Angular and React.js or mobile applications.
- *Razor Pages web applications*—Razor Pages applications generate HTML on the server and are designed to be viewed by users in a web browser directly.
- *MVC applications*—MVC applications are similar to Razor Pages apps in that they generate HTML on the server and are designed to be viewed by users directly in a web browser. They use traditional MVC controllers instead of Razor Pages.
- *Web API applications*—Web API applications are similar to minimal API apps, in that they are typically consumed by SPAs and mobile apps. Web API apps provide additional functionality compared to minimal APIs, at the expense of some performance and convenience.

We'll look at each of these application types in this book, but in part 1 we focus on minimal APIs, so in section 3.2.1 we start by looking at the simplest ASP.NET Core app you can create.

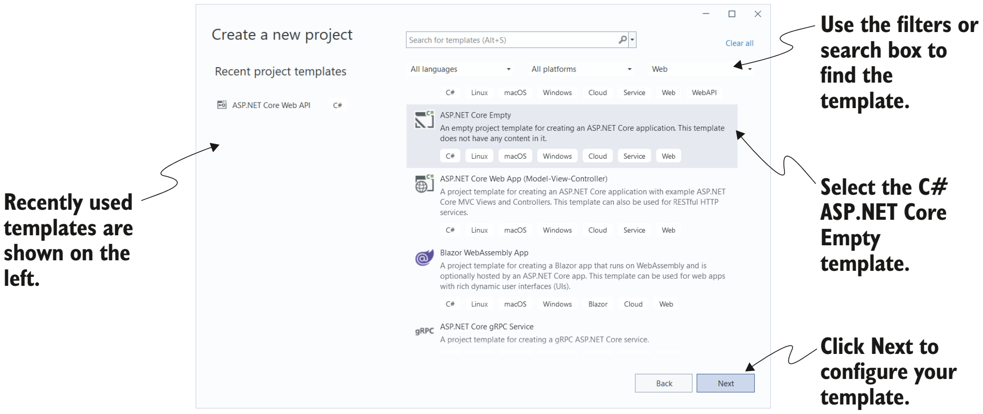
3.2.1 Using a template to get started

In this section you'll use a template to create your first ASP.NET Core minimal API application. Using a template can get you up and running with an application quickly, automatically configuring many of the fundamental pieces. Both Visual Studio and the .NET CLI come with standard templates for building web applications, console applications, and class libraries.

TIP In .NET, a *project* is a unit of deployment, which will be compiled into a .dll file or an executable, for example. Each separate app is a separate project. Multiple projects can be built and developed at the same time in a *solution*.

To create your first web application, open Visual Studio, and perform the following steps:

1. Choose **Create a New Project** from the splash screen, or choose **File > New > Project** from the main Visual Studio screen.
2. From the list of templates, choose **ASP.NET Core Empty**; select the C# language template, as shown in figure 3.2; and then choose **Next**.


Use the filters or search box to find the template.
Recently used templates are shown on the left.
Select the C# ASP.NET Core Empty template.
Click Next to configure your template.
Create a new project
Recent project templates
ASP.NET Core Web API C#
Search for templates (Alt+S)
Clear all
All languages
All platforms
Web
Back
Next

ASP.NET Core Empty
An empty project template for creating an ASP.NET Core application. This template does not have any content in it.
C# Linux macOS Windows Cloud Service Web

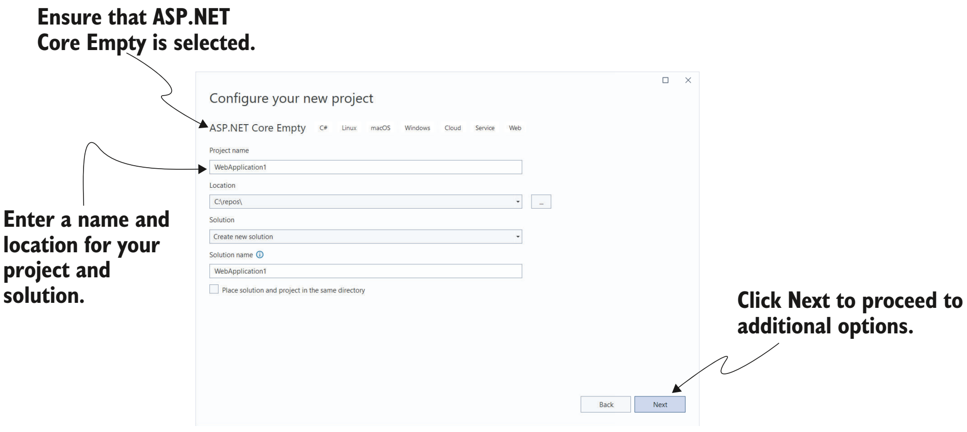
ASP.NET Core Web App (Model-View-Controller)
A project template for creating an ASP.NET Core application with example ASP.NET Core MVC Views and Controllers. This template can also be used for RESTful HTTP services.
C# Linux macOS Windows Cloud Service Web

Blazor WebAssembly App
A project template for creating a Blazor app that runs on WebAssembly and is optionally hosted by an ASP.NET Core app. This template can be used for web apps with rich dynamic user interfaces (UIs).
C# Linux macOS Windows Blazor Cloud Web

ASP.NET Core gRPC Service
A project template for creating a gRPC ASP.NET Core service.

Figure 3.2 The Create a New Project dialog box. Select the C# ASP.NET Core Empty template in the list on the right side. When you next create a new project, you can choose a template from the Recent Project Templates list on the left side.

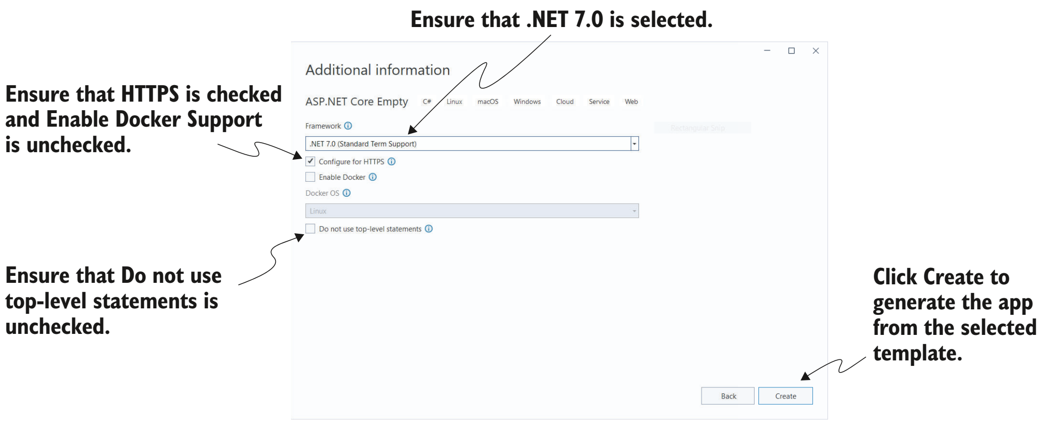
3. On the next screen, enter a project name, location, and solution name, and choose **Create**, as shown in figure 3.3. You might use WebApplication1 as both the project and solution name, for example.


Ensure that ASP.NET Core Empty is selected.
Enter a name and location for your project and solution.
Click Next to proceed to additional options.
Configure your new project
ASP.NET Core Empty C# Linux macOS Windows Cloud Service Web
Back
Next

Project name: WebApplication1
Location: C:\repos\
Solution: Create new solution
Solution name: WebApplication1
Place solution and project in the same directory

Figure 3.3 The Configure Your New Project dialog box. Enter a project name, location, and solution name, and choose Next.

4 On the following screen (figure 3.4), do the following:
   d Select **.NET 7.0**. If this option isn't available, ensure that you have .NET 7 installed. See appendix A for details on configuring your environment.
   e Ensure that **Configure for HTTPS** is checked.
   f Ensure that **Enable Docker** is not checked.
   g Ensure that **Do not use top-level statements** is not checked. (I explain top-level statements in section 3.6.)
   h Choose **Create**.


Ensure that .NET 7.0 is selected.
Ensure that HTTPS is checked and Enable Docker Support is unchecked.
Ensure that Do not use top-level statements is unchecked.
Click Create to generate the app from the selected template.
Additional information
ASP.NET Core Empty
C#
Linux
macOS
Windows
Cloud
Service
Web
Back
Create

Framework: .NET 7.0 (Standard Term Support)
[✓] Configure for HTTPS
[ ] Enable Docker
Docker OS: Linux
[ ] Do not use top-level statements

Figure 3.4 The Additional Information dialog box follows the Configure Your New Project dialog box and lets you customize the template that will generate your application. For this starter project, you'll create an empty .NET 7 application that uses top-level statements.

5 Wait for Visual Studio to generate the application from the template. When Visual Studio finishes, an introductory page about ASP.NET Core appears; you should see that Visual Studio has created and added some files to your project, as shown in figure 3.5.

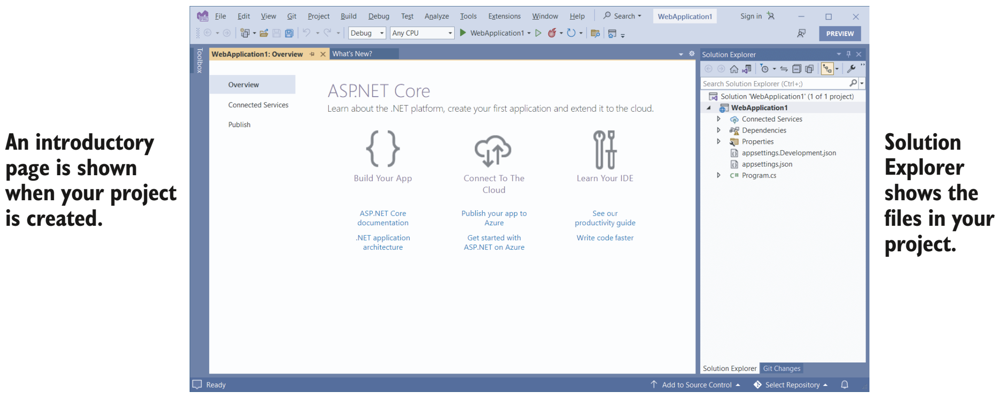

An introductory page is shown when your project is created.
Solution Explorer shows the files in your project.
File
Edit
View
Git
Project
Build
Debug
Test
Analyze
Tools
Extensions
Window
Help
Search
WebApplication1
Sign in
PREVIEW
Debug
Any CPU
WebApplication1: Overview
What's New?
Overview
Connected Services
Publish
ASP.NET Core
Learn about the .NET platform, create your first application and extend it to the cloud.
Build Your App
Connect To The Cloud
Learn Your IDE
ASP.NET Core documentation
.NET application architecture
Publish your app to Azure
Get started with ASP.NET on Azure
See our productivity guide
Write code faster
Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'WebApplication1' (1 of 1 project)
WebApplication1
Connected Services
Dependencies
Properties
appsettings.Development.json
appsettings.json
Program.cs
Git Changes
Ready
Add to Source Control
Select Repository

Figure 3.5 Visual Studio after creating a new ASP.NET Core application from a template. The Solution Explorer shows your newly created project. The introductory page has helpful links for learning about ASP.NET Core.

If you're not using Visual Studio, you can create a similar template by using the .NET CLI. Create a folder to hold your new project. Open a PowerShell or cmd prompt in the folder (Windows) or a terminal session (Linux or macOS), and run the commands in the following listing.

Listing 3.1 Creating a new minimal API application with the .NET CLI

```
dotnet new sln -n WebApplication1
dotnet new web -o WebApplication1
dotnet sln add WebApplication1
```

Creates a solution file called WebApplication1 in the current folder

Creates an empty ASP.NET Core project in a subfolder, WebApplication1

Adds the new project to the solution file

NOTE Visual Studio uses the concept of a solution to work with multiple projects. The example solution consists of a single project, which is listed in the .sln file. If you use a CLI template to create your project, you won't have a .sln file unless you generate it explicitly by using additional .NET CLI templates (listing 3.1).

Whether you use Visual Studio or the .NET CLI, now you have the basic files required to build and run your first ASP.NET Core application.

3.2.2 Building the application

At this point, you have most of the files necessary to run your application, but you've got two steps left. First, you need to ensure all the dependencies used by your project are downloaded to your machine, and second, you need to compile your application so that it can be run.

The first step isn't strictly necessary, as both Visual Studio and the .NET CLI automatically restore packages when they create your project, but it's good to know what's going on. In earlier versions of the .NET CLI, before 2.0, you needed to restore packages manually by using `dotnet restore`.

You can compile your application by choosing Build > Build Solution, pressing the shortcut Ctrl-Shift-B, or running `dotnet build` from the command line. If you build from Visual Studio, the output window shows the progress of the build, and assuming that everything is hunky-dory, Visual Studio compiles your application, ready for running. You can also run the `dotnet build` console commands from the Package Manager Console in Visual Studio.

TIP Visual Studio and the .NET CLI tools build your application automatically when you run it if they detect that a file has changed, so you generally won't need to perform this step explicitly yourself.

NuGet packages and the .NET CLI

One of the foundational components of .NET 7 cross-platform development is the .NET CLI, which provides several basic commands for creating, building, and running .NET 7 applications. Visual Studio effectively calls these commands automatically, but you can also invoke them directly from the command line if you're using a different editor. The most common commands used during development are

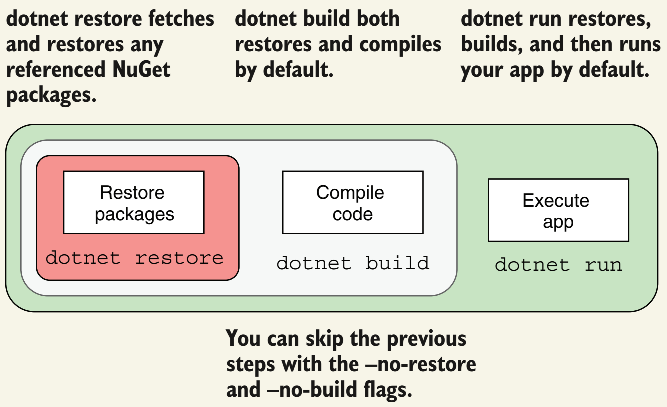
- `dotnet restore`
- `dotnet build`
- `dotnet run`

Each of these commands should be run inside your project folder and will act on that project alone. Except where explicitly noted, this is the case for all .NET CLI commands.

Most ASP.NET Core applications have dependencies on various external libraries, which are managed through the NuGet package manager. These dependencies are listed in the project, but the files of the libraries themselves aren't included. Before you can build and run your application, you need to ensure that there are local copies of each dependency on your machine. The first command, `dotnet restore`, ensures that your application's NuGet dependencies are downloaded and the files are referenced correctly by your project.

ASP.NET Core projects list their dependencies in the project's .csproj file, an XML file that lists each dependency as a `PackageReference` node. When you run `dotnet restore`, it uses this file to establish which NuGet packages to download. Any dependencies listed are available for use in your application.

The restore process typically happens implicitly when you build or run your application, as shown in the following figure, but it can be useful sometimes to run it explicitly, such as in continuous-integration build pipelines.


dotnet restore fetches and restores any referenced NuGet packages.
dotnet build both restores and compiles by default.
dotnet run restores, builds, and then runs your app by default.
Restore packages
dotnet restore
Compile code
dotnet build
Execute app
dotnet run
You can skip the previous steps with the –no-restore and –no-build flags.

The `dotnet build` command runs `dotnet restore` implicitly. Similarly, `dotnet run` runs `dotnet build` and `dotnet restore`. If you don't want to run the previous steps automatically, you can use the `--no-restore` and `--no-build` flags, as in `dotnet build --no-restore`.

You can compile your application by using `dotnet build`, which checks for any errors in your application and, if it finds no problems, produces output binaries that can be run with `dotnet run`.

Each command contains switches that can modify its behavior. To see the full list of available commands, run

```
dotnet --help
```

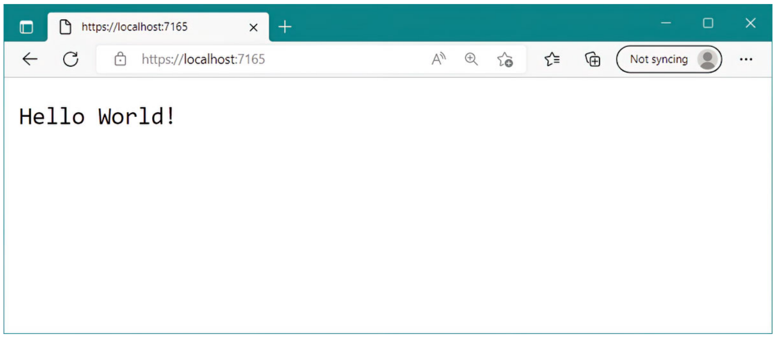
To see the options available for a particular command, such as `new`, run

```
dotnet new --help
```

3.3 Running the web application

You're ready to run your first application, and you have several ways to go about it. In Visual Studio, you can click the green arrow on the toolbar next to WebApplication1 or press the F5 shortcut. Visual Studio will automatically open a web browser window for you with the appropriate URL, and after a second or two, you should see the basic `"Hello World!"` response, as shown in figure 3.6.

Alternatively, instead of using Visual Studio, you can run the application from the command line with the .NET CLI tools by using `dotnet run`. Then you can open the URL in a web browser manually, using the address provided on the command line. Depending on whether you created your application with Visual Studio, you may see an http:// or https:// URL.



```
Hello World!
```

Figure 3.6 The output of your new ASP.NET Core application. The template chooses a random port to use for your application's URL, which will be opened in the browser automatically when you run from Visual Studio.

TIP The first time you run the application from Visual Studio, you may be prompted to install the development certificate. Doing so ensures that your browser doesn't display warnings about an invalid certificate.[2] See chapter 28 for more about HTTPS certificates.

[2] You can install the development certificate in Windows and macOS. For instructions on trusting the certificate on Linux, see your distribution's instructions. Not all browsers (Mozilla Firefox, for example) use the certificate store, so follow your browser's guidelines for trusting the certificate. If you still have difficulties, see the troubleshooting tips at http://mng.bz/o1pr.

This basic application has a single endpoint that returns the plain-text response when you request the path /, as you saw in figure 3.6. There isn't anything more you can do with this simple app, so let's look at some code!

3.4 Understanding the project layout

When you're new to a framework, creating an application from a template can be a mixed blessing. On one hand, you can get an application up and running quickly, with little input required on your part. Conversely, the number of files can be overwhelming, leaving you scratching your head working out where to start. The basic web application template doesn't contain a huge number of files and folders, as shown in figure 3.7, but I'll run through the major ones to get you oriented.

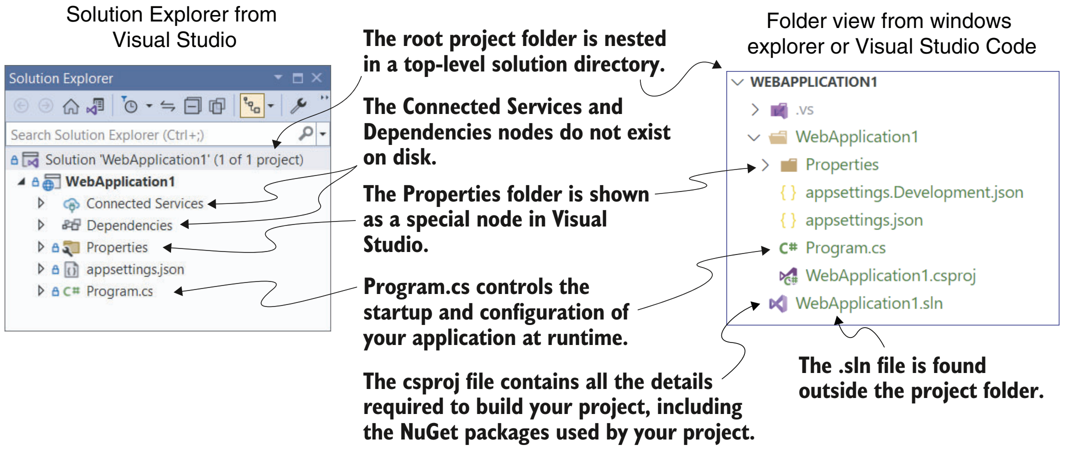

Solution Explorer from Visual Studio
Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'WebApplication1' (1 of 1 project)
WebApplication1
Connected Services
Dependencies
Properties
appsettings.json
Program.cs
The root project folder is nested in a top-level solution directory.
The Connected Services and Dependencies nodes do not exist on disk.
The Properties folder is shown as a special node in Visual Studio.
Program.cs controls the startup and configuration of your application at runtime.
The csproj file contains all the details required to build your project, including the NuGet packages used by your project.
Folder view from windows explorer or Visual Studio Code
WEBAPPLICATION1
.vs
WebApplication1
Properties
appsettings.Development.json
appsettings.json
Program.cs
WebApplication1.csproj
WebApplication1.sln
The .sln file is found outside the project folder.

Figure 3.7 Solution Explorer and folder on disk for a new ASP.NET Core application. Solution Explorer also displays the Connected Services and Dependencies nodes, which list NuGet and other dependencies, though the folders themselves don't exist on disk.

The first thing to notice is that the main project, WebApplication1, is nested in a top-level directory with the name of the solution, which is also WebApplication1 in this case. Within this top-level folder you'll also find the solution (.sln) file used by Visual Studio, though this is hidden in Visual Studio's Solution Explorer view.

Inside the solution folder you'll find your project folder, which contains the most important file in your project: WebApplication1.csproj. This file describes how to build your project and lists any additional NuGet packages that it requires. Visual Studio doesn't show the .csproj file explicitly, but you can edit it if you double-click the project name in Solution Explorer or right-click and choose Properties from the contextual menu. We'll take a closer look at this project file in the next section.

Your project folder contains a subfolder called Properties, which contains a single file: launchSettings.json. This file controls how Visual Studio will run and debug the application. Visual Studio shows the file as a special node in Solution Explorer, out of alphabetical order, near the top of your project. You've got two more special nodes in the project, Dependencies and Connected Services, but they don't have corresponding folders on disk. Instead, they show a collection of all the dependencies, such as NuGet packages, and remote services that the project relies on.

In the root of your project folder, you'll find two JSON files: appsettings.json and appsettings.Development.json. These files provide configuration settings that are used at runtime to control the behavior of your app.

Finally, Visual Studio shows one C# file in the project folder: Program.cs. In section 3.6 you'll see how this file configures and runs your application.

3.5 The .csproj project file: Declaring your dependencies

The .csproj file is the project file for .NET applications and contains the details required for the .NET tooling to build your project. It defines the type of project being built (web app, console app, or library), which platform the project targets (.NET Core 3.1, .NET 7 and so on), and which NuGet packages the project depends on.

The project file has been a mainstay of .NET applications, but in ASP.NET Core it has had a facelift to make it easier to read and edit. These changes include

- *No GUIDs*—Previously, globally unique identifiers (GUIDs) were used for many things, but now they're rarely used in the project file.
- *Implicit file includes*—Previously, every file in the project had to be listed in the .csproj file to be included in the build. Now files are compiled automatically.
- *No paths to NuGet package .dll files*—Previously, you had to include the path to the .dll files contained in NuGet packages in the .csproj, as well as list the dependencies in a packages.config file. Now you can reference the NuGet package directly in your .csproj, and you don't need to specify the path on disk.

All these changes combine to make the project file far more compact than you'll be used to from previous .NET projects. The following listing shows the entire .csproj file for your sample app.

Listing 3.2 The .csproj project file, showing SDK, target framework, and references

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>net7.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
  </PropertyGroup>
</Project>
```

The SDK attribute specifies the type of project you're building.

The TargetFramework is the framework you'll run on—in this case, .NET 7.

Enables the C# 8 feature "nullable reference types"

Enables the C# 10 feature "implicit using statements"

For simple applications, you probably won't need to change the project file much. The `Sdk` attribute on the `Project` element includes default settings that describe how to build your project, whereas the `TargetFramework` element describes the framework your application will run on. For .NET 6.0 projects, this element will have the `net6.0` value; if you're running on .NET 7, this will be `net7.0`. You can also enable and disable various features of the compiler, such as the C# 8 feature nullable reference types or the C# 10 feature implicit using statements.[3]

TIP With the new csproj style, Visual Studio users can double-click a project in Solution Explorer to edit the .csproj file without having to close the project first.

The most common changes you'll make to the project file are to add more NuGet packages by using the `PackageReference` element. By default, your app doesn't reference any NuGet packages at all.

Using NuGet libraries in your project

Even though all apps are unique in some way, they also have common requirements. Most apps need to access a database, for example, or manipulate JSON- or XML-formatted data. Rather than having to reinvent that code in every project, you should use existing reusable libraries.

NuGet is the library package manager for .NET, where libraries are packaged in *NuGet packages* and published to https://www.nuget.org. You can use these packages in your project by referencing the unique package name in your .csproj file, making the package's namespace and classes available in your code files. You can publish (and host) NuGet packages to repositories other than nuget.org; see https://learn.microsoft.com/en-us/nuget for details.

You can add a NuGet reference to your project by running `dotnet add package <packagename>` from inside the project folder. This command updates your project file with a `<PackageReference>` node and restores the NuGet package for your project. To install the popular Newtonsoft.Json library, for example, you would run

```
dotnet add package Newtonsoft.Json
```

This command adds a reference to the latest version of the library to your project file, as shown next, and makes the Newtonsoft.Json namespace available in your source-code files:

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>net7.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="NewtonSoft.Json" Version="13.0.1" />
  </ItemGroup>
</Project>
```

[3] You can read about the new C# features included in .NET 7 and C# 11 at http://mng.bz/nWMg.

If you're using Visual Studio, you can manage packages with the NuGet Package Manager by right-clicking the solution name or a project and choosing Manage NuGet Packages from the contextual menu.

As a point of interest, there's no officially agreed-on pronunciation for NuGet. Feel free to use the popular "noo-get" or "nugget" style, or if you're feeling especially posh, try "noo-jay"!

The simplified project file format is much easier to edit by hand than previous versions, which is great if you're developing cross-platform. But if you're using Visual Studio, don't feel that you have to take this route. You can still use the GUI to add project references, exclude files, manage NuGet packages, and so on. Visual Studio will update the project file itself, as it always has.

TIP For further details on the changes to the csproj format, see the documentation at http://mng.bz/vnzJ.

The project file defines everything Visual Studio and the .NET CLI need to build your app—everything, that is, except the code! In the next section we'll look at the file that defines your whole ASP.NET Core application: the Program.cs file.

3.6 Program.cs file: Defining your application

All ASP.NET Core applications start life as a .NET Console application. As of .NET 6, that typically means a program written with *top-level statements,* in which the startup code for your application is written directly in a file instead of inside a `static void Main` function.

Top-level statements

Before C# 9, every .NET program had to include a `static void Main` function (it could also return `int`, `Task`, or `Task<int>`), typically declared in a class called `Program`. This function, which must exist, defines the entry point for your program. This code runs when you start your application, as in this example:

```
using System;
namespace MyApp
{
    public class Program
    {
        public static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

With top-level statements you can write the body of this method directly in the file, and the compiler generates the `Main` method for you.

(continued)

When combined with C# 10 features such as implicit using statements, this dramatically simplifies the entry-point code of your app to

```
Console.WriteLine("Hello World!");
```

When you use the explicit `Main` function you can access the command-line arguments provided when the app was run using the `args` parameter. With top-level statements the `args` variable is also available as a `string[]`, even though it's not declared explicitly. You could echo each argument provided by using

```
foreach(string arg in args)
{
    Console.WriteLine(arg);
}
```

In .NET 7 all the default templates use top-level statements, and I use them throughout this book. Most of the templates include an option to use the explicit `Main` function if you prefer (using the `--use-program-main` option if you're using the CLI). For more information on top-level statements and their limitations, see http://mng.bz/4DZa. If you decide to switch approaches later, you can always add or remove the `Main` function manually as required.

In .NET 7 ASP.NET Core applications the top-level statements build and run a `WebApplication` instance, as shown in the following listing, which shows the default Program.cs file. The `WebApplication` is the core of your ASP.NET Core application, containing the application configuration and the Kestrel server that listens for requests and sends responses.

Listing 3.3 The default Program.cs file that configures and runs a `WebApplication`

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();

app.MapGet("/", () => "Hello World!");

app.Run();
```

Creates a WebApplicationBuilder using the CreateBuilder method

Builds and returns an instance of WebApplication from the WebApplicationBuilder

Defines an endpoint for your application, which returns Hello World! when the path "/" is called

Runs the WebApplication to start listening for requests and generating responses

These four lines contain all the initialization code you need to create a web server and start listening for requests. It uses a `WebApplicationBuilder`, created by the call to `CreateBuilder`, to define how the `WebApplication` is configured, before instantiating the `WebApplication` with a call to `Build()`.

NOTE You'll find this pattern of using a builder object to configure a complex object repeated throughout the ASP.NET Core framework. This technique is useful for allowing users to configure an object, delaying its creation until all configuration has finished. It's also one of the patterns described in the "Gang of Four" book *Design Patterns: Elements of Reusable Object-Oriented Software,* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addison-Wesley, 1994).

In this simple application we don't make any changes to `WebApplicationBuilder` before calling `Build()`, but `WebApplicationBuilder` configures a lot of things by default, including

- *Configuration*—Your app loads values from JSON files and environment variables that you can use to control the app's runtime behavior, such as loading connection strings for a database. You'll learn more about the configuration system in chapter 10.
- *Logging*—ASP.NET Core includes an extensible logging system for observability and debugging. I cover the logging system in detail in chapter 26.
- *Services*—Any classes that your application depends on for providing functionality—both those used by the framework and those specific to your application—must be registered so that they can be instantiated correctly at runtime. The `WebApplicationBuilder` configures the minimal set of services needed for an ASP.NET Core app. Chapters 8 and 9 look at service configuration in detail.
- *Hosting*—ASP.NET Core uses the Kestrel web server by default to handle requests.

After configuring the `WebApplicationBuilder` you call `Build()` to create a `WebApplication` instance. The `WebApplication` instance is where you define how your application handles and responds to requests, using two building blocks:

- *Middleware*—These small components execute in sequence when the application receives an HTTP request. They can perform a whole host of functions, such as logging, identifying the current user for a request, serving static files, and handling errors. We'll look in detail at the middleware pipeline in chapter 4.
- *Endpoints*—Endpoints define how the response should be generated for a specific request to a URL in your app.

For the application in listing 3.3, we didn't add any middleware, but we defined a single endpoint using a call to `MapGet`:

```
app.MapGet("/", () => "Hello World!");
```

You use the `MapGet` function to define how to handle a request that uses the `GET` *HTTP verb*. There are other `Map*` functions for other HTTP verbs, such as `MapPost`.

DEFINITION Every HTTP request includes a *verb* that indicates the type of the request. When you're browsing a website, the default verb is `GET`, which *fetches* a resource from the server so you can view it. The second-most-common verb is `POST`, which is used to *send* data to the server, such as when you're completing a form.

The first argument passed to `MapGet` defines which URL path to respond to, and the second argument defines *how* to generate the response as a delegate that returns a `string`. In this simple case, the arguments say "When a request is made to the path `/` using the `GET` *HTTP verb*, respond with the plain-text value `Hello World!`".

> **DEFINITION** A *path* is the remainder of the request URL after the domain has been removed. For a request to www.example.org/accout/manage, the path is `/account/manage`.

While you're configuring the `WebApplication` and `WebApplicationBuilder` the application isn't handling HTTP requests. Only after the call to `Run()` does the HTTP server start listening for requests. At this point, your application is fully operational and can respond to its first request from a remote browser.

> **NOTE** The `WebApplication` and `WebApplicationBuilder` classes were introduced in .NET 6. The initialization code in previous versions of ASP.NET Core was more verbose but gave you more control of your application's behavior. Configuration was typically split between two classes—`Program` and `Startup`—and used different configuration types—`IHostBuilder` and `IHost`, which have fewer defaults than `WebApplication`. In chapter 30 I describe some of these differences in more detail and show how to configure your application by using the generic `IHost` instead of `WebApplication`.

So far in this chapter, we've looked at the simplest ASP.NET core application you can build: a `Hello World` minimal API application. For the remainder of this chapter, we're going to build on this app to introduce some fundamental concepts of ASP.NET Core.

3.7 Adding functionality to your application

The application setup you've seen so far in Program.cs consists of only four lines of code but still shows the overall *structure* of a typical ASP.NET Core app entry point, which typically consists of six steps:

1. Create a `WebApplicationBuilder` instance.
2. Register the required services and configuration with the `WebApplicationBuilder`.
3. Call `Build()` on the builder instance to create a `WebApplication` instance.
4. Add middleware to the `WebApplication` to create a pipeline.
5. Map the endpoints in your application.
6. Call `Run()` on the `WebApplication` to start the server and handle requests.

The basic minimal API app shown previously in listing 3.3 was simple enough that it didn't need steps 2 and 4, but otherwise it followed this sequence in its Program.cs file. The following listing extends the default application to add more functionality, and in doing so it uses all six steps.

Listing 3.4 The Program.cs file for a more complex example minimal API

```
using Microsoft.AspNetCore.HttpLogging;

WebApplicationBuilder builder = WebApplication.CreateBuilder(args);

builder.Services.AddHttpLogging(opts =>
    opts.LoggingFields = HttpLoggingFields.RequestProperties);

builder.Logging.AddFilter(
    "Microsoft.AspNetCore.HttpLogging", LogLevel.Information);

WebApplication app = builder.Build();

if (app.Environment.IsDevelopment())
{
    app.UseHttpLogging();
}

app.MapGet("/", () => "Hello World!");
app.MapGet("/person", () => new Person("Andrew", "Lock"));

app.Run();

public record Person(string FirstName, string LastName);
```

You can customize features by adding or customizing the services of the application.

Ensures that logs added by the HTTP logging middleware are visible in the log output

You can add middleware conditionally, depending on the runtime environment.

The HTTP logging middleware logs each request to your application in the log output.

Creates a new endpoint that returns the C# object serialized as JSON

Creates a record type

The application in listing 3.4 configures two new features:

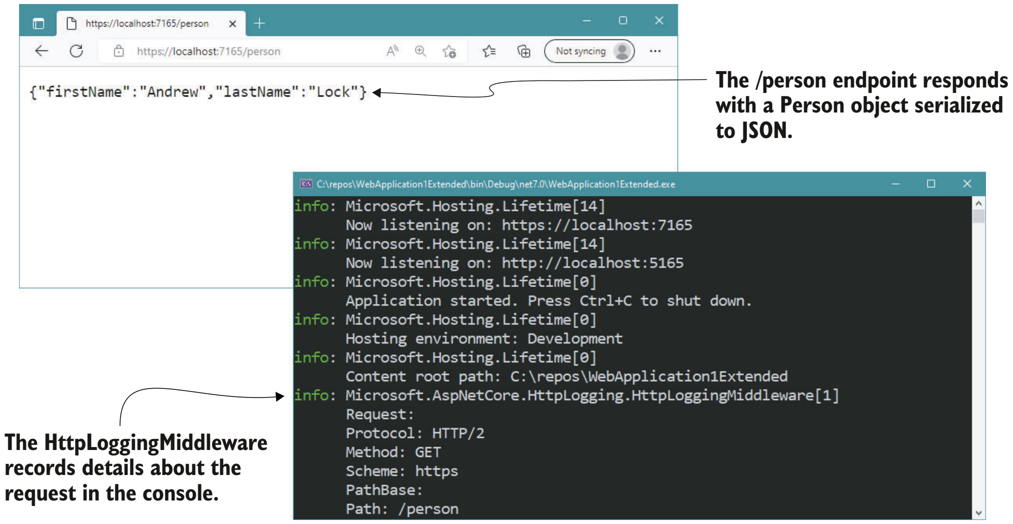
- When running in the Development environment, details about each request are logged using the `HttpLoggingMiddleware`.[4]
- Creates a new endpoint at `/person` that creates an instance of the C# record called `Person` and serializes it in the response as JSON.

When you run the application and send requests via a web browser, you see details about the request displayed in the console, as shown in figure 3.8. If you call the `/person` endpoint you'll see the JSON representation of the `Person` record you created in the endpoint.

NOTE You can view the application only on the same computer that's running it at the moment; your application isn't exposed to the internet yet. You'll learn how to publish and deploy your application in chapter 27.

Configuring services, logging, middleware, and endpoints is fundamental to building ASP.NET Core applications, so the rest of section 3.7 walks you through each of these concepts to give you a taste of how they're used. I won't explain them in detail (we have the rest of the book for that!), but you should keep in mind how they follow on from each other and how they contribute to the application's configuration as a whole.

[4] You can read in more detail about HTTP logging in the documentation at http://mng.bz/QPmw.


https://localhost:7165/person
Not syncing
C:\repos\WebApplication1Extended\bin\Debug\net7.0\WebApplication1Extended.exe
The /person endpoint responds with a Person object serialized to JSON.
The HttpLoggingMiddleware records details about the request in the console.

```
{"firstName":"Andrew","lastName":"Lock"}
```

```
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: https://localhost:7165
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:5165
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\repos\WebApplication1Extended
info: Microsoft.AspNetCore.HttpLogging.HttpLoggingMiddleware[1]
      Request:
      Protocol: HTTP/2
      Method: GET
      Scheme: https
      PathBase:
      Path: /person
```

Figure 3.8 Calling the `/person` endpoint returns a JSON-serialized version of the `Person` record instance. Details about each request are logged to the console by the `HttpLoggingMiddleware`.

3.7.1 Adding and configuring services

ASP.NET Core uses small modular components for each distinct feature. This approach allows individual features to evolve separately, with only a loose coupling to others, and it's generally considered to be good design practice. The downside to this approach is that it places the burden on the consumer of a feature to instantiate it correctly. Within your application, these modular components are exposed as one or more *services* that are used by the application.

DEFINITION Within the context of ASP.Net Core, *service* refers to any class that provides functionality to an application. Services could be classes exposed by a library or code you've written for your application.

In an e-commerce app, for example, you might have a `TaxCalculator` that calculates the tax due on a particular product, taking into account the user's location in the world. Or you might have a `ShippingCostService` that calculates the cost of shipping to a user's location. A third service, `OrderTotalCalculator`, might use both of these services to work out the total price the user must pay for an order. Each service provides a small piece of independent functionality, but you can combine them to create a complete application. This design methodology scenario is known as the *single-responsibility principle.*

DEFINITION The *single-responsibility principle* (SRP) states that every class should be responsible for only a single piece of functionality; it should need to change only if that required functionality changes. SRP is one of the five main design principles promoted by Robert C. Martin in *Agile Software Development, Principles, Patterns, and Practices* (Pearson, 2013).

`OrderTotalCalculator` needs access to an instance of `ShippingCostService` and `TaxCalculator`. A naive approach to this problem is to use the `new` keyword and create an instance of a service whenever you need it. Unfortunately, this approach tightly couples your code to the specific implementation you're using and can undo all the good you achieved by modularizing the features in the first place. In some cases, it may break the SRP by making you perform initialization code in addition to using the service you created.

One solution to this problem is to make it somebody else's problem. When writing a service, you can declare your dependencies and let another class fill those dependencies for you. Then your service can focus on the functionality for which it was designed instead of trying to work out how to build its dependencies.

This technique is called *dependency injection* or the *Inversion of Control* (IoC) principle, a well-recognized design pattern that is used extensively. Typically, you'll register the dependencies of your application into a *container,* which you can use to create any service. You can use the container to create both your own custom application services and the framework services used by ASP.NET Core. You must register each service with the container before using it in your application.

NOTE I describe the dependency inversion principle and the IoC container used in ASP.NET Core in detail in chapters 8 and 9.

In an ASP.NET Core application, this registration is performed by using the `Services` property of `WebApplicationBuilder`. Whenever you use a new ASP.NET Core feature in your application, you need to come back to Program.cs and add the necessary services. This task isn't always as arduous as it sounds, typically requiring only a line or two of code to configure your applications.

In listing 3.4 we configured an optional service for the HTTP logging middleware by using the line

```
builder.Services.AddHttpLogging(opts =>
    opts.LoggingFields = HttpLoggingFields.RequestProperties);
```

Calling `AddHttpLogging()` adds the necessary services for the HTTP logging middleware to the IoC container and customizes the options used by the middleware for what to display. `AddHttpLogging` isn't exposed directly on the `Services` property; it's an extension method that provides a convenient way to encapsulate all the code required to set up HTTP logging. This pattern of encapsulating setup behind extension methods is common in ASP.NET Core.

As well as registering framework-related services, the `Services` property is where you'd register any custom services you have in your application, such as the example `TaxCalculator` discussed previously. The `Services` property is an `IServiceCollection`, which is a list of every known service that your application will need to use. By adding a new service to it, you ensure that whenever a class declares a dependency on your service, the IoC container will know how to provide it.

As well as configuring services, `WebApplicationBuilder` is where you customize other cross-cutting concerns, such as logging. In listing 3.4, I showed how you can add a logging filter to ensure that the logs generated by the `HttpLoggingMiddleware` are written to the console:

```
builder.Logging.AddFilter(
    "Microsoft.AspNetCore.HttpLogging", LogLevel.Information);
```

This line ensures that logs of severity `Information` or greater created in the `Microsoft.AspNetCore.HttpLogging` namespace will be included in the log output.

> **NOTE** I show configuring log filters in code here for convenience, but this isn't the idiomatic approach for configuring filters in ASP.NET Core. Typically, you control which levels are shown by adding values to appsettings.json instead, as shown in the source code accompanying this chapter. You'll learn more about logging and log filtering in chapter 26.

After you call `Build()` on the `WebApplicationBuilder` instance, you can't register any more services or change your logging configuration; the services defined for the `WebApplication` instance are set in stone. The next step is defining how your application responds to HTTP requests.

3.7.2 Defining how requests are handled with middleware and endpoints

After registering your services with the IoC container on `WebApplicationBuilder` and doing any further customization, you create a `WebApplication` instance. You can do three main things with the `WebApplication` instance:

- Add middleware to the pipeline.
- Map endpoints that generate a response for a request.
- Run the application by calling `Run()`.

As I described previously, middleware consists of small components that execute in sequence when the application receives an HTTP request. They can perform a host of functions, such as logging, identifying the current user for a request, serving static files, and handling errors. Middleware is typically added to `WebApplication` by calling `Use*` extension methods. In listing 3.4, I showed an example of adding the `HttpLoggingMiddleware` to the middleware pipeline conditionally by calling `UseHttpLogging()`:

```
if (app.Environment.IsDevelopment())
{
    app.UseHttpLogging();
}
```

We added only a single piece of middleware to the pipeline in this example, but when you're adding multiple pieces of middleware, the order of the `Use*` calls is important: the order in which they're added to the builder is the order in which they'll execute in the final pipeline. Middleware can use only objects created by previous middleware in the pipeline; it can't access objects created by later middleware.

> **WARNING** It's important to consider the order of middleware when adding it to the pipeline, as middleware can use only objects created earlier in the pipeline.

You should also note that listing 3.4 uses the `WebApplication.Environment` property (an instance of `IWebHostEnvironment`) to provide different behavior when you're in a development environment. The `HttpLoggingMiddleware` is added to the pipeline only when you're running in development; when you're running in production (or, rather, when `EnvironmentName` is *not* set to `"Development"`), the `HttpLoggingMiddleware` will not be added.

> **NOTE** You'll learn about hosting environments and how to change the current environment in chapter 10.

The `WebApplicationBuilder` builds an `IWebHostEnvironment` object and sets it on the `Environment` property. `IWebHostEnvironment` exposes several environment-related properties, such as

- `ContentRootPath`—Location of the working directory for the app, typically the folder in which the application is running
- `WebRootPath`—Location of the wwwroot folder that contains static files
- `EnvironmentName`—Whether the current environment is a development or production environment

`IWebHostEnvironment` is already set by the time the `WebApplication` instance is created. `EnvironmentName` is typically set externally by using an environment variable when your application starts.

Listing 3.4 added only a single piece of middleware to the pipeline, but `WebApplication` *automatically* adds more middleware, including two of the most important and substantial pieces of middleware in the pipeline: the *routing* middleware and the *endpoint* middleware. The routing middleware is added automatically to the *start* of the pipeline, before any of the additional middleware added in Program.cs (so before the `HttpLoggingMiddleware`). The endpoint middleware is added to the *end* of the pipeline, after all the other middleware added in Program.cs.

> **NOTE** `WebApplication` adds several more pieces of middleware to the pipeline by default. It automatically adds error-handling middleware when you're running in the development environment, for example. I discuss some of this autoadded middleware in detail in chapter 4.

Together, this pair of middleware is responsible for interpreting the request to determine which endpoint to invoke, for reading parameters from the request, and for generating the final response. For each request, the *routing* middleware uses the request's URL to determine which endpoint to invoke. Then the rest of the middleware pipeline executes until the request reaches the endpoint middleware, at which point the endpoint middleware executes the endpoint to generate the final response.

The routing and endpoint middleware work in tandem, using the set of endpoints defined for your application. In listing 3.4 we defined two endpoints:

```
app.MapGet("/", () => "Hello World!");
app.MapGet("/person", () => new Person("Andrew", "Lock"));
```

You've already seen the default `"Hello World!"` endpoint. When you send a `GET` request to `/`, the routing middleware selects the `"Hello World!"` endpoint. The request continues down the middleware pipeline until it reaches the endpoint middleware, which executes the lambda and returns the `string` value in the response body.

The other endpoint defines a lambda to run for `GET` requests to the `/person` path, but it returns a C# record instead of a `string`. When you return a C# object from a minimal API endpoint, the object is serialized to JSON automatically and returned in the response body, as you saw in figure 3.8. In chapter 6 you'll learn how to customize this response, as well as return other types of responses.

And there you have it. You've finished the tour of your first ASP.NET Core application! Before we move on, let's take one last look at how our application handles a request. Figure 3.9 shows a request to the `/person` path being handled by the sample application. You've seen everything here already, so the process of handling a request should be familiar. The figure shows how the request passes through the middleware pipeline before being handled by the endpoint middleware. The endpoint executes the lambda method and generates the JSON response, which passes back through the middleware to the ASP.NET Core web server before being sent to the user's browser.

The trip has been pretty intense, but now you have a good overview of how an entire application is configured and how it handles a request by using minimal APIs. In chapter 4, you'll take a closer look at the middleware pipeline that exists in all ASP.NET Core applications. You'll learn how it's composed, how you can use it to add functionality to your application, and how you can use it to create simple HTTP services.

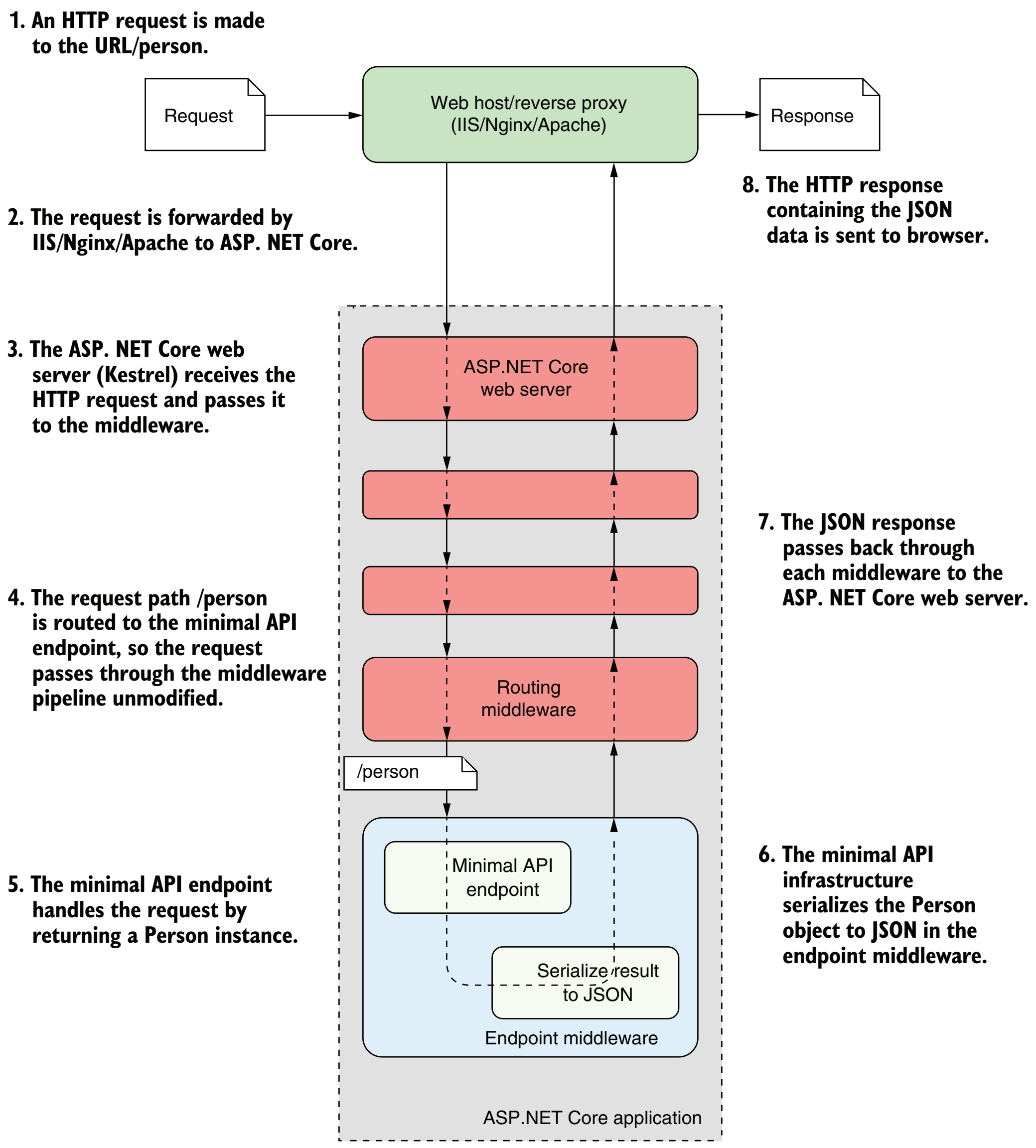

1. An HTTP request is made to the URL/person.
Request
Web host/reverse proxy (IIS/Nginx/Apache)
Response
8. The HTTP response containing the JSON data is sent to browser.
2. The request is forwarded by IIS/Nginx/Apache to ASP. NET Core.
3. The ASP. NET Core web server (Kestrel) receives the HTTP request and passes it to the middleware.
ASP.NET Core web server
7. The JSON response passes back through each middleware to the ASP. NET Core web server.
4. The request path /person is routed to the minimal API endpoint, so the request passes through the middleware pipeline unmodified.
Routing middleware
/person
5. The minimal API endpoint handles the request by returning a Person instance.
Minimal API endpoint
Serialize result to JSON
Endpoint middleware
6. The minimal API infrastructure serializes the Person object to JSON in the endpoint middleware.
ASP.NET Core application

Figure 3.9 An overview of a request to the `/person` URL for the extended ASP.NET Core minimal API application. The routing middleware routes the request to the correct lambda method. The endpoint generates a JSON response by executing the method and passes the response back through the middleware pipeline to the browser.

Summary

- The .csproj file contains the details of how to build your project, including which NuGet packages it depends on. Visual Studio and the .NET CLI use this file to build your application.
- Restoring the NuGet packages for an ASP.NET Core application downloads all your project's dependencies so that it can be built and run.
- Program.cs is where you define the code that runs when your app starts. You can create a `WebApplicationBuilder` by using `WebApplication.CreateBuilder()` and call methods on the builder to create your application.
- All services, both framework and custom application services, must be registered with the `WebApplicationBuilder` by means of the `Services` property, to be accessed later in your application.
- After your services are configured you call `Build()` on the `WebApplicationBuilder` instance to create a `WebApplication` instance. You use `WebApplication` to configure your app's middleware pipeline, to register the endpoints, and to start the server listening for requests.
- Middleware defines how your application responds to requests. The order in which middleware is registered defines the final order of the middleware pipeline for the application.
- The `WebApplication` instance automatically adds `RoutingMiddleware` to the start of the middleware pipeline and `EndpointMiddleware` as the last middleware in the pipeline.
- Endpoints define how a response should be generated for a given request and are typically tied to a request's path. With minimal APIs, a simple function is used to generate a response.
- You can start the web server and begin accepting HTTP requests by calling `Run` on the `WebApplication` instance.